CentMesh Drones Mission Control Protocol

Specifications, API, How-to

Objective: The objective of this document is to describe the protocol by which an application running on the CentMesh Mobile Node platform can receive high-level directives from a Mission Control. In this context, the term “Mission Manager” and “Mission Control Protocol” refer specifically to CentMesh Mobile Nodes and CentMesh Drones, and are not to be confused with MAVLink terms such as “Ground Control Station”. This document also provides an introduction to the MCP sample code provided (both for the drone side, and the controller side), and describes how the sample code can be utilized by a drone application programmer.

Change History:
Version 0.1: Created by Ramachandra Kasyap Marmavula, March 6, 2014
Version 0.2: Edited by Rudra Dutta: added overview section, March 6, 2014
Version 0.3: Edited by Ramachandra Kasyap Marmavula: added How-to section March 7, 2014

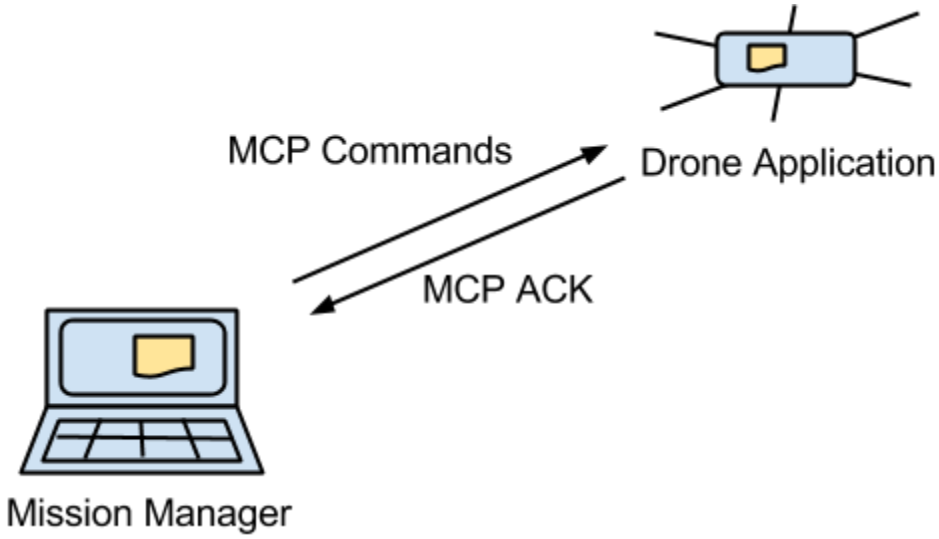
More Context: CentMesh website and wiki: http://centmesh.csc.ncsu.edu

---

1 Overview of the Mission Control Protocol (MCP)

The MCP is primarily intended to allow applications running on CentMesh Drones to receive mission control commands from a “Mission Manager”.

The Mission Manager and drone application communicate over UDP sockets, by convention on ports 7889 and 7890. At the IP level, they are local subnet IP broadcasts. MCP messages themselves can be unicast or broadcast. Unicast MCP messages should be replied to with an MCP acknowledgement message.



The format of the messages exchanged in the MCP is as below:

Preamble	Dst ID	Src ID	Sequence No	Action Code	Payload

The **Preamble** is a 9-byte unique pattern. The first 7 bytes spell out “NCSU-CM” in ASCII, the value of each of the last two bytes is 1.

The next two fields provide the MCP ID of the destination and source of the message. By convention, the Mission Manager always has the ID 0. The drones each have a unique ID locally configured, which can be read from a file by (or specified on the command line of) the drone application.

The **Sequence No** is a 2-byte field that allows successive messages to be identified and acknowledged independently. Each sender of MCP messages (i.e, Mission Manager, or Drone Application) starts with an arbitrary initial sequence number value, and then increments it for successive messages. Retransmissions of the same message are identified by repeating sequence number. This allows a sender to guarantee delivery of a unicast messages by retransmitting the message until an acknowledgement is received. The receiver discards duplicates received for a message, if any.

The 2-byte **Action Code** field specifies the meaning of the MCP message, and the **Payload** field provides the details for the action code. The length and meaning of the **Payload** is dependent on the **Action Code**.

The following values are possible for the **Action Code** field, with corresponding meanings. ALL of the following messages except ACK and Mission_Ready are always sent by the Mission Manager. An ACK is sent by the drone application or Mission Manager for every unicast message received.

Mission_Ready: The drone application sends this message to the Mission Manager signaling its 'mission readiness'. The drone application must verify GPS lock and other similar ready-to-fly capabilities before sending this message. The Mission Manager waits for a 'Mission_Ready' message from each drone application before it proceeds with initiating the mission.

Start_Mission: The Mission Manager starts the mission with this message, and does not send any subsequent MCP commands until an acknowledgement is received. The drone application must not ARM the autopilot before receiving this message. No payload.

End_Mission: The Mission Manager sends this message to indicate that it will send no further commands in the current mission. The drone application should complete carrying out all currently pending actions (from previously received commands), then execute an RTL. No payload.

WP_List: A list of waypoints is embedded in the payload. The first two bytes of the payload specify the number of waypoints that follow. The waypoints are listed in the Waypoint File format. The drone application should interpret the waypoints in a way appropriate to the mission. (For example, in Challenge 1, this should be interpreted as the waypoints to visit, but not necessarily in that order.)

WP_On: A single waypoint is embedded in the payload, in MAVLink format, and is now considered "active". The meaning of "active" depends on the mission. (For example, in Challenge 2, it means a waypoint is now eligible for earning points.)

WP_Off: The most recently specified waypoint to be "active" is now "inactive". The meaning of "inactive" depends on the mission. No payload.

Obstacles_List: A list of waypoints is embedded in the payload, that should be interpreted as obstacles. The first two bytes of the payload specify the number of waypoints that follow. The obstacles are specified in Waypoint File format.
(Please note that the Waypoint File format is recommended by MAVLink, though not part of MAVLink and the same format is used by MCP. Sample waypoint and the way of extraction is described in a later section in this document).

ACK: The payload is a sequence number for a previously received message, which is being acknowledged. This is the only message sent by the drone application.

Mav_Passthrough: This code is reserved for future compatibility. The payload consists of a single embedded MAVLink message.

2 MCP Application Programmer's Interface

In this section we describe an API to perform the various necessary tasks for the MCP Mission Manager and Drone Application programs. The API is made available as a python library. In the first section below, we describe the classes with their data members. In the next section, we describe the methods available in these classes. The last section describes how an application programmer can use these methods to communicate using the MCP protocol, and perform the various common required tasks.

2.1 Class Definitions

The code for the MCP implementation consists of two key components:

1. MCP module for sending and receiving MCP messages
2. Custom modules (to be written by application developers) that uses (1) to handle received messages. There are default handlers for each and every message that do 'nothing'. In other words, if the application developers do not write any message handlers, no action will be taken whenever a MCP message is received.

File: mcp.py

```
==========================================================
from mcp_common import Action_codes

# class that is offered by the MCP module, used to send and receive messages
class Mcp()
{
        # Attributes
        udp_server_port = None #  Local server that is created in the constructor
        remote_udp_conn = None # Tuple with (Broadcast IP, remote port)
        mcp_handler = None # This is replaced by the one called in the constructor
        timeout =         5 # in seconds, can be overridden by what is passed
        num_of_retries=1000 # can be overridden by the passed argument
        preamble = ASCII code corresponding to N' 'C' 'S' 'U' '-' 'C' 'M' followed by 0x01 0x01
        source_ID = None # This will be initialized in the constructor, for now it is 1 for UAV and
                                #0 for laptop
        base_format_string = "!7s2B4H" #This has to be appended by the format for payload

        """ Function List """
        # constructor
        __init__(self, handler_obj (of type mission_message_handler:
        mandatory),port_to_listen_to : optional, port_to_send_to: optional, IP address to send to:
```

mandatory, source_id: mandatory):

destructor
__del__(self):

Public function: to send unicast message.
#Exception with the appropriate error message is thrown when the function fails.
If timeout and retries are not specified the call blocks.

send_unicast_message (action_code , buffer, destination ID (all mandatory), timeout, num of retries: optional):

Public function
send_broadcast_message (action_code, buffer):

“”” Public function: if the message is destined for itself, ACK, otherwise no ACKing. Calls the registered handlers. If an ACK is received, ignore and continue listening for a‘proper’ message.
OPTIONAL: can specify what sort of message to wait for and timeout. If timeout is not specified, the call blocks until a message of expected type is received.
”””
recv_message (message_type = <one of the action codes>, timeout = 0)

“”” Private function: Function that actually sends messages - both send_broadcast_message() and send_unicast_message() call this. This method does the packing of the structure according to what is expected. The handling of retries and timeouts is handled by the wrapper send_() functions that invoke this function. “”” __send__ (payload, destination ID)

“”” Private function: Function that actually receives messages - this is called by recv_message() and also send_() functions to wait for an ACK.”””
__recv__(timeout)

“”” Private function: Returns a string that has the packet in the MCP format. Uses the format string and packs the structure.
format_string = base_format_string + len(payload) + s”””
__generate_mcp_packet__ (action_code, buffer):

"""Private function: Parses the buffer received and returns the following:
    source_id, seq_id, action_code, payload"""
__get_mcp_fields__(self, buffer):

```
        “””Private function: Increment the sequence number. Called during every send””.
        __increment_sequence_number__ ():

        """Private function: Returns the length of MCP Header. Length of current fields:
        Preamble: 'NCSU-CM'0x010x01 (9 bytes)
        destination ID: 2 bytes
        source ID: 2 bytes
        sequence number: 2 bytes
        Action code: 2 bytes
        Total: 17 bytes
        """
        __length_of_mcp_header__():

        “"" Private function: Return the current sequence number"""
        __get_sequence_number__ ():

        “””Private function: returns the destination broadcast ID”””
        __get_dest_broadcast_ID__():

        “”” Private function: Validate if the received action code is a valid action code. Returns
        True if valid, False if not”””
        __validate_action_code__ (action_code):

        “”” Private function: Check if the passed action code is what was expected. This is called
        by recv_message (action_code = something). Returns True or False.“””
        __check_passed_action_code__ (action_code):

        """ Private function: Get remote end address. Used by send()."""
        __get_remote_end_address__(self):


}

# Class for generating exceptions.
class McpException(Exception):
{
# Constructor: takes the message to be used as argument
__init__(message):

# Returns the message that was used to create this object
__str__():

}
```

File: mcp_enum.py
==========

“””
Reference: http://stackoverflow.com/questions/36932/how-can-i-represent-an-enum-in-python
“””

“””To check if a given action code exists. we can simply call Action_code.reverse_mapping(<value>) in a try/catch block. If this returns a key error, then the action code is invalid.
“””

```
def enum(*sequential, **named):
    enums = dict(zip(sequential, range(len(sequential))), **named)
    reverse = dict((value, key) for key, value in enums.iteritems())
    enums['reverse_mapping'] = reverse
    return type('Enum', (), enums)
```

File: mcp_common.py
==========

```
import mcp_enum

"""
Current Action codes:

Mission_Ready = 0
Start_Mission = 1
End_Mission = 2
WP_List = 3
WP_On = 4
WP_Off = 5
Obstacles_List = 6
ACK = 7
Mav_Passthrough = 8
NOTE: 'Any' type is used on the receiving/sending end for validation purpose
only. It can't be used as an actual action code.
"""
Action_Codes = mcp_enum.enum('Mission_Ready','Start_Mission','End_Mission','WP_List',
'WP_On','WP_Off', 'Obstacles_List', 'ACK','Mav_Passthrough','Any')
```

File: mcp_handler.py

```
=======================
from mcp_common import Action_codes

class MissionMessageHandlerBase(Object) {
        # Attributes
        dict_message_handlers = {}
        mavutil_obj = None # We need this object to control the UAV.

        “”” Function List “””
        # constructor
        __init__(mavutil_obj (of type mavutil: mandatory - should allow None as well as we would
        not need one for the process running on the Laptop)):

        # destructor
        __del__():

        # Public: Function to handle Start_mission. Starts the mission by arming the copter
        handle_start_mission_message():

        # Public: Function to handle End_mission. Should probably do a RTL.
        handle_end_mission_message():

        # Public: Function to handle WP_List. Saves the waypoints to a dictionary - [index, WP]
        handle_wp_list_message(buffer):

        # Public: Function to handle WP_On. Save the waypoint and perform some action.
        handle_wp_on_message(buffer):

        # Public: Function to handle WP_Off. Mark the waypoint and perform some action.
        handle_wp_off_message(buffer):

        # Public: Function to handle obstacle list
        handle_obstacle_list_message (buffer):

        # Public: Function to handle Mav_Passthrough. Custom action.
        handle_mav_passthrough_message (buffer):

        # No handler required for ACK message and we don’t call this either. But adding this
        for the sake of completion:
        handle_ack_message():
```

```
        # The handlers MUST be overridden by their exact names in any class that extends this.
        # Preferring a simple (but a bit difficult to follow) method for overriding.


        #Public func: NOT a message handler. This function 'blocks' till the APM is 'mission
        #ready'. We currently verify if there is a GPS fix and wait till we have one. More
        #functionality can be possibly added here
        is_mission_ready():
}
```

2.2 Available methods - Sample code description

File: mission_manager.py

Process running on the laptop that manages the UAV missions

==========

This has the main() method that does the following making use of mcp module:

1. Waits for 'Mission Ready' messages from all UAVs (currently only one)
2. Sends 'Start Mission' message to each UAV
3. Executes the challenge based on the challenge number

File present in ‘mission_manager’ directory.

Invoke it using the following syntax:

./mission_manager.py <challenge #> <broadcast address> <local port> <remote port>

Example: For challenge 2 and testing locally (i.e. both Mission Manager and the drone application reside in the same system), use the following command:

./mission_manager.py 2 127.255.255.255 7889 7890

File: mcp_handler_ground.py

Has the method definitions for handling MCP messages for the Mission Manager

==========

As mentioned in the earlier section, there are default message handlers for each MCP message (that do nothing). If a new handler is not defined for a particular function, then the default one is invoked. Since the Mission Manager handles only ‘Mission_Ready’ message, this file has handler just for the single message:

File present in ‘mission_manager’ directory.

class MissionMessageHandler (a sub-class of MissionMessageHandlerBase):

```
# This keeps track of whether the UAVs are mission ready or not
dict_is_uav_ready = {}
#Constructor: initialize the dictionary with list of system IDs
__init__(mavutil_obj,list_system_ids):

#Public: Handler for 'Mission_Ready' message. This is the only message
#that the mission planner is interested in. The buffer contains the ID of
#UAV that sent this message"""
handle_mission_ready_message (buffer):

#Public: Returns true if all UAVs are 'mission ready'. False otherwise
all_uavs_ready(self):
```

File: mission_controller.py

Process running on the UAV that is managed by Mission Manager

==========

This has the main() method that does the following making use of mcp module:

1. Sends 'Mission Ready' message to mission planner (after getting a GPS fix)
2. Waits for 'Start Mission' message from mission planner. Upon receipt, ARMs the UAV and makes it fly vertically to an altitude of 20 metres
3. Waits for 'Waypoint List' message (challenge 1), saves the waypoints to a dictionary and then prints out (index of waypoint, latitude, longitude, altitude) for each waypoint.

File present in the following path: ‘sample_prog_mavutil/program_3a/mission_controller.py’

Invoke it using the following syntax:

./mcp_mission_controller.py <broadcast address> <local port> <remote port>

Example: For testing locally (i.e. both Mission Manager and the drone application reside in the same system), use the following command:

./mcp_mission_controller.py 127.255.255.255 7890 7889

Note that the values specified for local port and remote port should be interchanged for Mission Manager and drone application.

File: mcp_handler_mission_controller.py

Has the method definitions for handling MCP messages for sample app 3a - Mission controller

==========

As mentioned in the earlier section, there are default message handlers for each MCP message (that do nothing). If a new handler is not defined for a particular function, then the default one is invoked. The sample application (mission controllers) handles ‘Start_Mission’,’End_Mission’ and ‘WP_List’ messages and handlers are defined for them.

(NOTE: In the sample code provided, there are also handlers for WP_On, WP_Off messages. Both the handlers print a waypoint after receiving one. Additional functionality (which will be needed for the challenges) can be built on top of this.

File present in the following path: ‘sample_prog_mavutil/program_3a/mcp_handler_uav.py’

class MissionMessageHandler (a sub-class of MissionMessageHandlerBase):

```
    # constructor
    __init__(mavutil_obj):

    # Handler for 'Start_Mission': ARM the UAV and station keep at 10 metres
    handle_start_mission_message(buffer):

    #Handler for 'End_Mission': The 'End_Mission' implies that the mission
    #manager will not send any more messages. The UAV can continue doing
    #whatever it is currently doing and then Return to Launch (RTL) once
    #that is done. Since this is sample code, we just do a RTL
    handle_end_mission_message(self, buffer = None):

    #Handler for WP_On: Displays the received waypoint.
    handle_wp_on_message(buffer):

    #Handler for WP_Off: Displays the received waypoint.
    handle_wp_off_message (buffer):

    #Handler for 'WP_List': this sample code saves the waypoints to a list and displays them
    handle_wp_list_message(buffer):

    #Internal func: NOT a message handle. Traverses the dictionary of waypoints saved
    #by the handler and prints the following: index of waypoint, latitude, longitude and
    #altitude
    __print_waypoints__():

    #Internal function: makes the UAV fly vertically up to 20 metres
    __fly_vertically__ ():
```

Extracting WayPoint list

As given in the Waypoint File format, following are the various fields:

<INDEX> <CURRENT WP> <COORD FRAME> <COMMAND> <PARAM1> <PARAM2> <PARAM3> <PARAM4>
<PARAM5/X/LONGITUDE> <PARAM6/Y/LATITUDE> <PARAM7/Z/ALTITUDE> <AUTOCONTINUE>

Following is the content of sample file is used in Challenge 1:

```
QGC WPL 110
0	1	0	16	0	0	0	0	35.771332	-78.674421
0.000000	1
1	0	3	22	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	3.000000	1
2	0	3	19	5.000000	0.000000	0.000000	0.000000
35.771103	-78.674659	3.000000	1
3	0	3	19	5.000000	0.000000	0.000000	0.000000
35.771057	-78.674421	3.000000	1
4	0	3	20	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	1
```

A waypoint list is built from this. The format of the list is as follows:

# of waypoints	Waypoint 1	Waypoint 2	…….	Waypoint n

<---2 bytes---> List of waypoints each of size 37 bytes

The size 37 comes from the fact that mavlink_apm.py uses the following method to 'pack'/'unpack' waypoints (reference 'pack()' method of 'MAVLink_mission_item_message' class:

```
MAVLink_message.pack(self, mav, 254, struct.pack('<fffffffHHBBBBB', self.param1,
self.param2, self.param3, self.param4, self.x, self.y, self.z, self.seq, self.command,
self.target_system, self.target_component, self.frame, self.current, self.autocontinue))
```

The sample app (mcp_mission_controller.py) blocks for 'WP_List' message after receiving a '*Start_Mission'* message. Once this message is received, '*handle_wp_list_message()*' in 'mcp_handler_uav.py' is invoked.

```
mcp_obj.recv_message(mcp.Action_Codes.WP_List)
```

Following is the list of steps performed in '*handle_wp_list_message()*' to obtain <latitude, longitude,altitude> for each waypoint:

1) Get the number of waypoints

```
wp_list_size = len(buffer) - 2
format_string = mcp.format_num_waypoints + str(wp_list_size) + 's'
num_waypoints, rest_of_payload = struct.unpack(format_string,buffer)
```

2) Save waypoints to a dictionary

```
wp_list = struct.unpack(format_string_wp,rest_of_payload)
idx = 0
for way_point in wp_list:
    self.dict_waypoints[idx] = way_point
    idx = idx+1
```

3) Print the index, latitude, longitude and altitude using the function '*__print_waypoints__()'.*

```
for key,value in self.dict_waypoints.items():
    waypoint_tuple = struct.unpack(fmt, value)
    print waypoint_tuple
    print ('Waypoint (%d) : Lat: %f Lon:%f Alt: %f'
            %(int(waypoint_tuple[7]),float(waypoint_tuple[4]),
            float(waypoint_tuple[5]),
            float(waypoint_tuple[6]))
```

Please note that there is a slight difference in the order in which elements are stored from the order in which they are packed and sent by mavlink_apm.py - in the Waypoint File format, latitude, longitude and altitude are present in the 'end':

```
<INDEX> <CURRENT WP> <COORD FRAME> <COMMAND> <PARAM1> <PARAM2> <PARAM3>
<PARAM4> <PARAM5/X/LONGITUDE> <PARAM6/Y/LATITUDE> <PARAM7/Z/ALTITUDE>
<AUTOCONTINUE>
```

mavlink_apm.py packs this and sends them in the format mentioned above where the latitude, longitude and altitude appear at indices 4,5 and 6 when treated as a tuple.

```
param1,param2,param3,param4,x,y,z,eq,command,target_system,target_component,frame,
current,autocontinue))
```

You are encouraged to use this example as a reference to extract waypoints and obstacle list

2.3 Using the Methods to design drone applications

The sample code with the above classes and methods can be easily utilized in building drone applications. The Mission Manager code can be used as is for the three challenges (use the first command line argument for specifying the particular challenge).

For developing the **drone application**, there is only one major component that needs to be written - the message handler for a message. For example, if the drone application requires to handle waypoint list, have a function handle_waypoint_list_message(buffer) and specify the functionality there. Note the name of the function should match the one specified in mcp.py.

For the general sequence of things to take care, refer the existing sample application (a combination of mcp_mission_controller.py and mcp_handler_uav.py). In short the flow would be this:

Handler module

In a new file, have a class that extends 'MissionMessageHandlerBase' class (ex: 'MissionMessageHandler'). This class should have methods for handling various messages that are expected to be received. 'Start_Mission' and 'End_Mission' are to be handled by all drone applications and thus should always have handlers.Which others to have depends on the role of the drone application (ex: for challenge 1,' WP_List' must be handled, while for challenge 2, 'WP_On' & 'WP_Off' must be handled etc.)

Main module

In another file, have the 'main' function that does the following (in that order):

1. Create a mavutil object:

    mavutil_obj = mavutil.mavlink_connection(device)

2. Create an object of type 'MissionMessageHandler' pasing mavutil_obj created above:

    mission_message_handler_obj = MissionMessageHandler(mavutil_obj)

3. Create an object of type mcp passing the 'mission_message_handler_obj',local port, remote port and broadcast address as arguments:

    mcp_obj = mcp.Mcp(mission_message_handler_obj,udp_local_port,udp_remote_port, broadcast_address, source_id)

4. Wait for the UAV to get 'mission ready' (in other words, call the below function and wait for it to return):

mcp_obj.mcp_handler_obj.is_mission_ready()

5. Send ‘Mission_Ready’ message to Mission Manager after (4) returns:
mcp_obj.send_unicast_message(mcp.Action_Codes.Mission_Ready,None,0)

6. Wait for Start_Mission message. On receiving one, ARM the UAV and station keep at 10 metres:
mcp_obj.recv_message(mcp.Action_Codes.Start_Mission)

7. Keep receiving messages using recv_message(). Once a message is received, the handler specified is automatically invoked (the application developer does not need to do anything).

8. Stop receiving messages once End_Mission is received. You can have a flag in the code to keep track of whether an End_Mission has been received. Upon ‘End_Mission’ receipt, complete the flight and do RTL.

NOTE:

- send_unicast_message() takes timeout, number of retries as optional arguments. If these are not specified, the call blocks.
- recv_message() takes timeout and expected action code as optional argument. If timeout is not specified, the call blocks. If a particular action code is specified and a message with some other action code is received, then the received message is ignored and the call blocks till the expected message is received.